Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery
(Extended Abstract)

Shlomi Dolev[1,⋆], Omri Liba[1], and Elad M. Schiller[2,⋆⋆]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
{dolev,liba}@cs.bgu.ac.il
[2] Department of Computer Science and Engineering,
Chalmers University of Technology, Goeteborg, Sweden
elad@chalmers.se

Abstract. Traditional Byzantine resilient algorithms use $2f + 1$ vertex-disjoint paths to ensure message delivery in the presence of up to f Byzantine nodes. The question of how these paths are identified is related to the fundamental problem of topology discovery. Distributed algorithms for topology discovery cope with a never ending task: dealing with frequent changes in the network topology and unpredictable transient faults. Therefore, algorithms for topology discovery should be self-stabilizing to ensure convergence of the topology information following any such unpredictable sequence of events. We present the first such algorithm that can cope with Byzantine nodes. Starting in an arbitrary global state, and in the presence of f Byzantine nodes, each node is eventually aware of all the other non-Byzantine nodes and their connecting communication links. Using the topology information, nodes can, for example, route messages across the network and deliver messages from one end user to another. We present the first deterministic, cryptographic-assumptions-free, self-stabilizing, Byzantine-resilient algorithms for network topology discovery and end-to-end message delivery. We also consider the task of r-neighborhood discovery for the case in which r and the degree of nodes are bounded by constants. The use of r-neighborhood discovery facilitates polynomial time, communication and space solutions for the above tasks. The obtained algorithms can be used to authenticate parties, in particular during the establishment of private secrets, thus forming public key schemes that are resistant to man-in-the-middle attacks of the compromised Byzantine nodes. A polynomial and efficient end-to-end algorithm that is based on the established private secrets can be employed in between periodical secret re-establishments.

⋆ Partially supported by Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences, Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium, Grant from the Institute for Future Defense Technologies Research named for the Medvedi of the Technion, and Israeli Internet Association.

⋆⋆ Partially supported by the EC, through project FP7-STREP-288195, KARYON (Kernel-based ARchitecture for bsafetY-critical cONtrol), the European Commission Seventh Framework Programme (FP7/2007-2013) under grant agreement 257007 and through the FP7-SEC-285477-CRISALIS project.

V. Gramoli and R. Guerraoui (Eds.): NETYS 2013, LNCS 7853, pp. 42–57, 2013.

1 Introduction

Self-stabilizing Byzantine resilient topology discovery is a fundamental distributed task that enables communication among parties in the network even if some of the components are compromised by an adversary. Currently, such topology discovery is becoming extremely important where countries' main infrastructures, such as the electrical smart-grid, water supply networks and intelligent transportation systems are subject to cyber-attacks. Self-stabilizing Byzantine resilient algorithms naturally cope with mobile attacks [e.g., 1]. Whenever the set of compromised components is fixed (or dynamic, but small) during a period that suffices for convergence of the algorithm, the system starts demonstrating useful behavior following the convergence. For example, consider the case in which nodes of the smart-grid are constantly compromised by an adversary while local recovery techniques, such as local node reset and/or refresh, ensure the recovery of a compromised node after a bounded time. Once the current compromised set does not imply a partition of the communication graph, the distributed control of the smart grid automatically recovers. Self-stabilizing Byzantine resilient algorithms for topology discovery and message delivery are important for systems that have to cope with unanticipated transient violations of the assumptions that the algorithms are based upon, such as unanticipated violation of the upper number of compromised nodes and unanticipated transmission interferences that is beyond the error correction code capabilities.

The dynamic and difficult-to-predict nature of electrical smart-grid and intelligent transportation systems give rise to many fault-tolerance issues and require efficient solutions. Such networks are subject to transient faults due to hardware/software temporal malfunctions or short-lived violations of the assumed settings for the location and state of their nodes. Fault-tolerant systems that are *self-stabilizing* [2] can recover after the occurrence of transient faults, which can drive the system to an arbitrary system state. The system designers consider *all* configurations as possible configurations from which the system is started. The self-stabilization design criteria liberate the system designer from dealing with specific fault scenarios, risking neglecting some scenarios, and having to address each fault scenario separately.

We also consider Byzantine faults that address the possibility of a node to be compromised by an adversary and/or to run a corrupted program, rather than merely assuming that they start in an arbitrary local state. Byzantine components may behave arbitrarily (selfishly, or even maliciously) as message senders and as relaying nodes. E.g., Byzantine nodes may block messages, selectively omit messages, redirect message routes, playback messages, or modify messages. Any system behavior is possible, when all (or one third or more of) the nodes are Byzantine nodes. Thus, the number of Byzantine nodes, f, is usually restricted to be less than one third of the nodes [2, 3].

The task of *r-neighborhood network discovery* allows each node to know the set of nodes that are at most r hops away from it in the communication network. Moreover, the task provides information about the communication links attached to these nodes. The task *topology discovery* considers knowledge regarding the node's entire connected component. The r-neighborhood network discovery and network topology discovery tasks are identical when r is the communication graph radius.

This work presents the first deterministic self-stabilizing algorithms for r-neighborhood discovery in the presence of Byzantine nodes. We assume that every r-neighborhood cannot be partitioned by the Byzantine nodes. In particular, we assume the existence of at least $2f + 1$ vertex-disjoint paths in the r-neighborhood, between any two non-Byzantine nodes, where at most f Byzantine nodes are present in the r-neighborhood, rather than in the entire network. [1] Note that by the self-stabilizing nature of our algorithms, recovery is guaranteed after a temporal violation of the above assumption. When r is defined to be the communication graph radius, our assumptions are equivalent to the standard assumption for Byzantine agreement in general (rather than only complete) communication graphs. In particular the standard assumption is that $2f + 1$ vertex disjoint paths exist and *are known* (see e.g., [3]) while we present distributed algorithms to find these paths starting in an arbitrary state.

Related Work. Self-stabilizing algorithms for finding vertex-disjoint paths for at most two paths between any pair of nodes, and for all vertex-disjoint paths in anonymous mesh networks appear in [4] and in [5], respectively. We propose self-stabilizing Byzantine resilient procedures for finding $f + 1$ vertex-disjoint paths in $2f + 1$-connected graphs. In [6], the authors study the problem of spanning tree construction in the presence of Byzantine nodes. Nesterenko and Tixeuil [7] presented preliminary ideas for a *non-stabilizing* algorithm for topology discovery in the presence of Byzantine nodes. Awerbuch and Sipser [8] consider algorithms that were designed for synchronous static network and give topology update as an example. They show how to use such algorithms in asynchronous dynamic networks. Unfortunately, their scheme starts from a consistent state and cannot cope with transient faults or Byzantine nodes.

The problems of *Byzantine gossip* [9–14] and *Byzantine Broadcast* [15, 16] consider the dissemination of information in the presence of Byzantine nodes rather than self-stabilizing topology discovery. Non-self-stabilizing Byzantine resilient gossip in the presence of one selfish node is considered in [10, 12]. In [11] the authors study oblivious deterministic gossip algorithms for multi-channel radio networks with a malicious adversary. They assume that the adversary can disrupt one channel per round, preventing communication on that channel. In [13] the authors consider probabilistic gossip mechanisms for reducing the redundant transmissions of flooding algorithms. They present several protocols that exploit local connectivity to adaptively correct propagation failures and protect against Byzantine attacks. Probabilistic gossip mechanisms in the context of recommendations and social networks are considered in [14]. In [9] the authors consider rules for avoiding a combinatorial explosion in (non-self-stabilizing) gossip protocol. Note that deterministic and self-stabilizing solutions are not presented in [9–14]. Drabkin et al. [15] consider non-self-stabilizing broadcast protocols that overcome Byzantine failures by using digital signatures, message signature gossiping, and failure detectors. Our deterministic self-stabilizing algorithm merely use the topological properties of the communication graph to ensure correct message delivery to the application layer in the presence of message omission, modifications and Byzantine nodes.

[1] Section 4 considers cases in which r and an upper bound on the node degree, Δ, are constants. For these cases, we have $\mathcal{O}(n)$ disjoint r-neighborhoods. Each of these (disjoint) r-neighborhoods may have up to f Byzantine nodes, and yet the above assumptions about at least $2f + 1$ vertex-disjoint paths in the r-neighborhood, hold.

A non-self-stabilizing broadcasting algorithm is considered in [16]. The authors assume the restricted case in which links and nodes of a communication network are subject to Byzantine failures, and that faults are distributed randomly and independently. We note that our result can serve as a base for a compiler that convert non-stabilizing algorithm to a stabilizing algorithm. We facilitate communication among participants that enables (repeatedly) run of a non-stabilizing algorithm that copes with Byzantine processors, using the standard re-synchronization technique that is based on self-stabilizing Byzantine clock synchronization [2, 17].

Our Contribution. We present two cryptographic-assumptions-free yet secure algorithms that are deterministic, self-stabilizing and Byzantine resilient.

We start by showing the existence of deterministic, self-stabilizing, Byzantine resilient algorithms for network topology discovery and end-to-end message delivery. The algorithms convergence time is in $\mathcal{O}(n)$. They take in to account every possible path and requiring bounded (yet exponential) memory and bounded (yet exponential) communication costs. Therefore, we also consider the task of r-neighborhood discovery, where r is a constant. We assume that if the r-neighborhood of a node has f Byzantine nodes, there are $2f+1$ vertex independent paths between the node and any non-Byzantine node in its r-neighborhood. The obtained r-neighborhood discovery algorithm requires polynomial memory and communication costs and supports deterministic, self-stabilizing, Byzantine-resilient algorithm for end-to-end message delivery across the network. Unlike topology update, the proposed end-to-end message delivery algorithm establishes message exchange synchronization between end-users that is based on message reception acknowledgments. Detailed proofs appear in [18].

2 Preliminaries

We consider settings of a standard asynchronous system [cf. 2]. The system consists of a set, $N = \{p_i\}$, of communicating entities, chosen from a set, P, which we call *nodes*. The upper bound on the number of nodes in the system is $n = |P|$. Each node has a unique identifier. Sometime we refer to a set, $P \setminus N$, of nonexisting nodes that a false indication on their existence can be recorded in the system. A node p_i can directly communicate with its *neighbors*, $N_i \subseteq N$. The system can be represented by an undirected network of directly communicating nodes, $G = (N, E)$, named the *communication graph*, where $E = \{(p_i, p_j) \in N \times N : p_j \in N_i\}$. We denote N_k's set of indices by $indices(N_k) = \{m : p_m \in N_k\}$ and N_k's set of edges by $edges(N_j) = \{p_j\} \times N_j$.

The r-neighborhood of a node $p_i \in N$ is the connected component that includes p_i and all nodes that can be reached from p_i by a path of length r or less. The problem of r-neighborhood topology discovery considers communication graphs in which p_i's degree, δ_i, is bounded by a constant Δ. Hence, when the neighborhood radius, r, and the node degree, Δ, are both constants the number of nodes in the r-neighborhood is also bounded by a constant, namely by $\mathcal{O}(\Delta^{r+1})$.

We model the communication channel, $queue_{i,j}$, from node p_i to node $p_j \in N_i$ as a FIFO queuing list of the messages that p_i has sent to p_j and p_j is about to receive. When p_i sends message m, the operation send() inserts a copy of m to the queue $queue_{i,j}$ of the one destination p_j, such that $p_j \in N_i$. We assume that the number of messages in

transit, i.e., stored in $queue_{i,j}$, is at most $capacity$. Once m arrives, p_j executes receive and m is dequeued.

We assume that p_i is completely aware of N_i, as in [7]. In particular, we assume that the identity of the sending node is known to the receiving one. In the context of the studied problem, we say that node $p_i \in N$ is *correct* if it reports on its genuine neighborhood, N_i. A *Byzantine* node, $p_b \in N$, is a node that can send arbitrarily corrupted messages. Byzantine nodes can introduce new messages and modify or omit messages that pass through them. This way they can, e.g., disinform correct nodes about their neighborhoods, about the neighborhood of other correct nodes, or the path through which messages travel, to name a very few specific misleading actions that Byzantine nodes may exhibit. Note that our assumptions do not restrict system settings in which a *duplicitous Byzantine* node, p_b, reports about N_b differently to its correct neighbors. In particular, p_b can have $\{N_{b_1}, \ldots N_{b_{\delta_b}}\}$ reports, such that p_b's identity in N_{b_i} is different than the one in N_{b_j}, where δ_x is the degree of node p_x. One may use a set of non-duplicitous Byzantine nodes, $\{p_{b_1}, \ldots p_{b_\delta}\}$, to model each of p_b's reports. Thus, for a $2k+1$ connected graph, the system tolerates no more than $\lfloor k/\Delta \rfloor$ duplicitous Byzantine nodes, where Δ is an upper bound on the node degree.

We denote C and B to be, respectively, the set of correct and Byzantine nodes. We assume that $|B| = f$, the identity of B's nodes is unknown to the ones in C, and B is fixed throughout the considered execution segment. These execution segments are long enough for convergence and then for obtaining sufficient useful work. We assume that between any pair of correct nodes there are at least $2f+1$ vertex-disjoints paths. We denote by $G_c = (C, E \cap C \times C)$ the *correct graph* induced by the set of correct nodes.

Self-stabilizing algorithms never terminate [2]. The non-termination property can be easily identified in the code of a self-stabilizing algorithm: the code is usually a do forever loop that contains communication operations with the neighbors. An iteration is said to be complete if it starts in the loop's first line and ends at the last (regardless of whether it enters branches).

Every node, p_i, executes a program that is a sequence of *(atomic) steps*. For ease of description, we assume the interleaving model with atomic step execution; a single step at any given time. An input event can either be the receipt of a message or a periodic timer going off triggering p_i to send. Note that the system is totally asynchronous and the (non-fixed) node processing rates are irrelevant to the correctness proof.

The *state* s_i of a node p_i consists of the value of all the variables of the node (including the set of all incoming communication channels, $\{queue_{j,i} | p_j \in N_i\}$. The execution of a step in the algorithm can change the state of a node. The term *(system) configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_n)$, where each s_i is the state of node p_i (including messages in transit for p_i). We define an *execution* $E = c[0], a[0], c[1], a[1], \ldots$ as an alternating sequence of system configurations $c[x]$ and steps $a[x]$, such that each configuration $c[x+1]$ (except the initial configuration $c[0]$) is obtained from the preceding configuration $c[x]$ by the execution of the step $a[x]$. We often associate the notation of a step with its executing node p_i using a subscript, e.g., a_i. An execution R (run) is *fair* if every correct node, $p_i \in C$, executes a step infinitely often in R. Time (e.g. needed for convergence) is measured by the number of *asynchronous rounds*, where the first asynchronous round is the minimal prefix of the

execution in which every node takes at least one step. The second asynchronous round is the first asynchronous round in the suffix of the run that follows the first asynchronous round, and so on. The message complexity (e.g. needed for convergence) is the number of messages measured in the specific case of synchronous execution.

We define the system's task by a set of executions called *legal executions* (LE) in which the task's requirements hold. A configuration c is a *safe configuration* for an algorithm and the task of LE provided that any execution that starts in c is a legal execution (belongs to LE). An algorithm is *self-stabilizing* with relation to the task LE when every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

3 Topology Discovery

The algorithm learns about the neighborhoods that the nodes report. Each report message contains an ordered list of nodes it passed so far, starting in a source node. These lists are used for verifying that the reports are sent over $f + 1$ vertex-disjoint paths.

When a report message, m, arrives to p_i, it inserts m to the queue $informedTopology_i$, and tests the queue consistency until there is enough independent evidence to support the report. The consistency test of p_i iterates over each node p_k such that, p_k appears in at least one of the messages stored in $informedTopology_i$. For each such node p_k, node p_i checks whether there are at least $f + 1$ messages from the same source node that have mutually vertex-disjoint paths and report on the same neighborhood. The neighborhood of each such p_k, that has at least $f + 1$ vertex-disjoint paths with identical neighborhood, is stored in the array $Result_i[k]$ and the total number of paths that relayed this neighborhood is kept in $Count[k]$.

We note that there may still be nodes $p_{fake} \in P \setminus (N)$, for which there is an entry $Result[fake]$. For example, $informedTopology$ may contain f messages, all originated from different Byzantine nodes, and a message m' that appears in the initial configuration and supports the (false) neighborhood the Byzantine messages refer to. These $f + 1$ messages can contain mutually vertex-disjoint paths, and thus during the consistency test, a result will be found for $Result[fake]$. We show that during the next computations, the message m' will be identified and ignored. The $Result$ array should include two reports for each (undirected) edge; the two nodes that are attached to the edge, each send a report. Hence, $Result$ includes a set of directed (report) edges. The term *contradicting edge* is needed when examining the $Result$ set consistency.

Definition 1 (Contradicting edges). *Given two nodes, $p_i, p_j \in P$, we say that the edge (p_i, p_j) is* contradicting with the set $evidence \subseteq edges(N_j)$, *if $(p_i, p_j) \notin evidence$.*

Following the consistency test, p_i examines the $Result$ array for contradictions. Node p_i checks the path of each message $m \in informedTopology_i$ with source p_r, neighborhood $neighborhood_r$ and $Path_r$. If every edge (p_s, p_j) on the path appears in $Result[s]$ and $Result[j]$, then we move to the next message. Otherwise, we found a fake supporter, and therefore we reduce $Count[r]$ by one. If the resulting $Count[r]$ is smaller than $f + 1$, we nullify the r'th entry of the $Result$ array. Once all messages are processed, the $Result$ array consisting of the (confirmed) local topologies is the output. At the end, p_i forwards the arriving message, m, to each neighbor that does not appear

in the path of m. The message sent by p_i includes the node from which m arrived as part of the visited path contained within m.

The Pseudocode of Algorithm 1. In every iteration of the infinite loop, p_i starts to compute its preliminary topology view by calling $ComputeResults$ in line 2. Then, every node p_k in the queue $InformedTopology$, node p_i goes over the messages in the queue from head to bottom. While iterating the queue, for every message m with source p_k, neighborhood N_k and visited path $Path_k$, p_i inserts $Path_k$ to $opinion[N_k]$, see line 18. After inserting, p_i checks if there is a neighborhood $Neig_k$ for which $opinion[Neig_k]$ contains at least $f + 1$ vertex-disjoint paths, see line 19. When such a neighborhood is found, it is stored in the $Result$ array (line 19). In line 20, p_i stores the number of vertex disjoint paths relayed messages that contained the selected neighborhood for p_k. After computing an initial topology view (line 3), p_i removes non-existing nodes from the computed topology. For every message m in $InformedTopology$, node p_i aims at validating its visited path. In line 24, p_i checks if there exists a node p_k whose neighborhood contradicts the visited path of m. If such a node exists, p_i decreases the associated entry in the $Count$ array (line 25). This decrease may cause $Count[r]$ to be smaller than $f + 1$, in this case p_i considers p_k to be fake and deletes the local topology of p_k from $Result[r]$ (line 26). Upon receiving a message m, node p_i inserts the message to the queue, in case it does not already exist, and just moves it to the queue top in case it does. The node p_i now needs to relay the message p_i got to all neighbors that are not on the message visited path (line 9). When sending, p_i also attaches the node identifier, from which the message was received, to the message visited path.

- $Insert(m)$: Insert item m to the queue head.
- $Remove(Messagem)$: Remove item m from the queue.
- $Iterator()$: Returns an pointer for iterating over the queue's items by their residence order in the queue.
- $HasNext()$: Tests whether the Iterator is at the queue end.
- $Next()$ Returns the next element to iterate over.
- $SizeOf()$ Returns the number of elements in the calling set.
- $MoveToHead(m)$: Move item m to the queue head.
- $IsAfter(m, S)$: Test that item m is after the items $m' \in S$, where S is the queue item set.

Fig. 1. $Queue$: general purpose data structure for queuing items, and its operation list

Algorithm's Correctness Proof. We now prove that within a linear amount of asynchronous rounds, the system stabilizes and every output is legal. The proof considers an arbitrary starting configuration with arbitrary messages in transit that could be actually in the communication channel or already stored in p_j's message queue and will be forwarded in the next steps of p_j. Each message in transit that traverses correct nodes can be forwarded within less than $\mathcal{O}(|C|)$ asynchronous rounds. Note that any message that traverses Byzantine nodes and arrives to a correct node that has at least one Byzantine node in its path. The reason is that the correct neighbor to the last Byzantine in the path lists the Byzantine node when forwarding the message. Thus, f is at most the number of messages that encode vertex-disjoint paths from a certain source that are initiated or corrupted by a Byzantine node. Since there are at least $f + 1$ vertex-disjoint paths with no Byzantine nodes from any source p_k to any node p_i and since p_k repeatedly sends messages to all nodes on all possible paths, p_i receives at least $f + 1$ messages from p_k with vertex-disjoint paths.

Algorithm 1. Topology discovery (code for node p_i)

Input: $Neighborhood_i$: The ids of the nodes with which node p_i can communicate directly;
Output: $ConfirmedTopology \subset P \times P$: Discovered topology, which is represent by a directed edge set;
Variable $InformedTopology : Queue$, see Figure 1: topological messages, $\langle node, neighborhood, path \rangle$;
Function: $NodeDisjointPaths(S)$: Test $S = \{\langle node, neighborhood, path \rangle\}$ to encode at least $f + 1$ vertex-disjoint paths;
Function: $PathContradictsNeighborhood(k, Neighborhood_k, path)$: Test that there is no node $p_j \in N$ for which there is an edge (p_k, p_j) in the message's visited path, $path \subseteq P \times N$, such that (p_k, p_j) is contradicting with $Neighborhood_k$;

1 **while true do**
2 $\quad Result \leftarrow ComputeResults()$
3 $\quad$ **let** $Result \leftarrow RemoveContradictions(Result)$
4 $\quad RemoveGarbage(Result)$
5 $\quad ConfirmedTopology \leftarrow ConfirmedTopology \cup (\bigcup_{p_k \in P} Result[k])$
6 $\quad$ **foreach** $p_k \in N_i$ **do send**$(i, Neighborhood_i, \emptyset)$ **to** p_k

7 **Upon Receive** $(\langle \ell, Neighborhood_\ell, VisitedPath_\ell \rangle)$ **from** p_j;
begin
8 $\quad Insert(p_\ell, Neighborhood_\ell, VisitedPath_\ell \cup \{j\})$
9 $\quad$ **foreach** $p_k \in N_i$ **do if** $k \notin VisitedPath_\ell$ **then send**$(p_\ell, Neighborhood_\ell, VisitedPath_\ell \cup \{j\})$ **to** p_k

10 **Procedure:** $Insert(k, Neighborhood_k, VisitedPath_k)$;
begin
11 $\quad$ **if** $\langle k, Neighborhood_k, VisitedPath_k \rangle \in InformedTopology$ **then** $InformedTopology.MoveToHead(m)$
12 $\quad$ **else if** $p_k \in N \wedge Neighborhood_k \subseteq indices(N) \wedge VisitedPath_k \subseteq indices(N)$ **then** $InformedTopology.Insert(\langle k, Neighborhood_k, VisitedPath_k \rangle)$

13 **Function:** $ComputeResults()$;
begin
14 $\quad$ **foreach** $p_k \in P : \langle k, Neighborhood_k, VisitedPath_k \rangle \in InformedTopology$ **do**
15 $\quad\quad$ **let** $(FirstDisjointPathsFound, Message, opinion[]) \leftarrow (false, InformedTopology.Iterator(), [\emptyset])$
16 $\quad\quad$ **while** $Message.hasNext()$ **do**
17 $\quad\quad\quad \langle \ell, Neighborhood_\ell, VisitedPath_\ell \rangle \leftarrow Message.Next()$
18 $\quad\quad\quad$ **if** $\ell = k$ **then** $opinion[Neighborhood_\ell].Insert(\langle \ell, Neighborhood_\ell, VisitedPath_\ell \rangle)$
19 $\quad\quad\quad$ **if** $FirstDisjointPathsFound = false \wedge NodeDisjointPaths(opinion[Neighborhood_\ell])$ **then** $(Result[k], FirstDisjointPathsFound) \leftarrow (Neighborhood_\ell, \mathbf{true})$
20 $\quad\quad Count[k] \leftarrow opinion[k][Result[k].SizeOf()$
21 $\quad$ **return** $Result$

22 **Function:** $RemoveContradictions(Result)$;
begin
23 $\quad$ **foreach** $\langle r, Neighborhood_r, VisitedPath_r \rangle \in InformedTopology$ **do**
24 $\quad\quad$ **if** $\exists p_k \in P : PathContradictsNeighborhood(p_k, Result[k], VisitedPath_r) = \mathbf{true}$ **then**
25 $\quad\quad\quad$ **if** $Neighborhood_r = Result[r]$ **then** $Count[r] \leftarrow Count[r] - 1$
26 $\quad\quad\quad$ **if** $Count[r] \leq f$ **then** $Result[r] \leftarrow \emptyset$
27 $\quad$ **return** $Result$

28 **Procedure:** $RemoveGarbage(Result)$;
begin
29 $\quad$ **foreach** $p_k \in N$ **do**
30 $\quad\quad$ **foreach** $m = \langle k, Neighborhood_k, VisitedPath_k \rangle \in InformedTopology : \{k\} \cup Neighborhood_k \cup VisitedPath_k \not\subseteq P \vee InformedTopology.IsAfter(m, opinion[k][Result[k]])$ **do** $InformedTopology.Remove(m)$

The FIFO queue usage and the repeated send operations of p_k ensure that the most recent $f + 1$ messages with vertex-disjoint paths in $InformedTopology$ queue are

uncorrupted messages. Namely, misleading messages that were present in the initial configuration will be pushed to appear below the new $f + 1$ uncorrupted messages. Thus, each node p_i eventually has the local topology of each correct node (stored in the $Result_i$ array). The opposite is however not correct as local topologies of non-existing nodes may still appear in the result array. For example, $InformedTopology_i$ may include in the first configuration $f + 1$ messages with vertex-disjoint paths for a non-existing node. Since after $ComputeResults$ we know the correct neighborhood of each correct node p_k, we may try to ensure the validity of all messages. For every message that encodes a non-existing source node, there must be a node p_ℓ on the message path, such that p_ℓ is correct and p_ℓ's neighbor is non-existing, this is true since p_i itself is correct. Thus, we may identify these messages and ignore them. Furthermore, no valid messages are ignored because of this validity check.

We also note that, since we assume that the nodes of the system are a subset of P, the size of the queue $InformedTopology$ is bounded. Lemma 1 bounds the needed amount of node memory (the proof details appear in [18]).

Lemma 1 (Bounded memory). *At any time, there are at most $n \cdot 2^{2n}$ messages in $InformedTopology_i$, where $p_i \in C$, $n = |P|$ and $\mathcal{O}(n \log(n))$ is the message size.*

r-neighborhood Discovery. Algorithm 1 demonstrates the existence of a deterministic self-stabilizing Byzantine resilient algorithm for topology discovery. Lemma 1 shows that the memory costs are high when the entire system topology is to be discovered. We note that one may consider the task of r-neighborhood discovery. Recall that in the r-neighborhood discovery task, it is assumed that every r-neighborhood cannot be partitioned by Byzantine nodes. Therefore, it is sufficient to constrain the maximal path length in line 9. The correctness proof of the algorithm for the r-neighborhood discovery follows similar arguments to the correctness proof of Algorithm 1.

4 End-to-End Delivery

We present a design for a self-stabilizing Byzantine resilient algorithm for the transport layer protocol that uses the output of Algorithm 1. The design is based on a function (named $getDisjointPaths()$) for selecting vertex-disjoint paths that contain a set of $f+1$ correct vertex-disjoint paths. We use $getDisjointPaths()$ and ARQ (Automatic Repeat reQuest) techniques for designing Algorithm 2, which ensures safe delivery between sender and receiver.

Exchanging Messages Over $f + 1$ Correct Vertex-Disjoint Paths. We guarantee correct message exchange by sending messages over a polynomial number of vertex-disjoint paths between the sender and the receiver. We consider a set, $CorrectPaths$, that includes $f + 1$ correct vertex-disjoint paths. Suppose that $ConfirmedTopology$ (see the output of Algorithm 1) encodes a set, $Paths$, of $2f+1$ vertex-disjoint paths between the sender and the receiver. It can be shown that $Paths$ includes at most f incorrect paths that each contains at least one Byzantine node, i.e., $Paths \supseteq CorrectPaths$. As we see next, $ConfirmedTopology$ does not always encode $Paths$, thus, one needs to circumvent this difficultly.

The case of constant r and Δ. The sender and the receiver exchange messages by using all possible paths between them; feasible considering r-neighborhoods, where the neighborhood radius, r, and the node degree Δ are constants.

The case of constant f. For each possible choice of f system nodes, $p_1, p_2, \ldots p_f$, the sender and the reciter compute a new graph $G(p_1, p_2, \ldots p_f)$ that is the result of removing $p_1, p_2, \ldots p_f$, from G_{out}, which is the graph defined by the discovered topology, $ConfirmedTopology$. Let $\mathcal{P}(p_1, p_2, \ldots p_f)$ be a set of $f + 1$ vertex-disjoint paths in $G(p_1, p_2, \ldots p_f)$ (or the empty set when $\mathcal{P}(p_1, p_2, \ldots p_f)$ does not exists) and $Paths = \bigcup_{p_1, p_2, \ldots p_f} \mathcal{P}(p_1, p_2, \ldots p_f)$. The sender and the receiver can exchange messages over $Paths$, because $|Paths|$ is polynomial at least one choice of $p_1, p_2, \ldots p_f$, has a corresponding set $\mathcal{P}(p_1, p_2, \ldots p_f)$ that contains $CorrectPaths$, see [18].

The case of no Byzantine neighbors The procedure assumes that any Byzantine node has no directly connected Byzantine neighbor in the communication graph. Specifically, this polynomial cost solution considers the (extended) graph, G_{ext}, that includes all the edges in $confirmedTopology$ and *suspicious edges*. Given three nodes, $p_i, p_j, p_k \in P$, we say that node p_i considers the undirected edge (p_k, p_j) suspicious, if the edge appears as a directed edge in $ConfirmedTopology_i$ for only one direction, e.g., (p_j, p_k).

The extended graph, G_{ext}, may contain fake edges that do not exists in the communication graph, but Byzantine nodes reports on their existence. Nevertheless, G_{ext} includes all the correct paths of the communication graph, G. Therefore, the $2f + 1$ vertex-disjoint paths that exists in G also exists in G_{ext} and they can facilitate a polynomial cost solution for the message exchange task, see [18].

Fig. 2. Implementation proposals for the function $getDisjointPaths()$

Note that even though $2f + 1$ vertex-disjoint paths between the sender and the receiver are present in the communication graph, the discovered topology in $ConfirmedTopology$ may not encode the set $Paths$, because f of the paths in the set $Paths$ can be controlled by Byzantine nodes. Namely, the information about at least one edge in f of the paths in the set $Paths$, can be missing in $ConfirmedTopology$.

We consider the problem of relaying messages over the set $CorrectPaths$ when only $ConfirmedTopology$ is known, and propose three implementations to the function $getDisjointPaths()$ in Figure 2. The value of $ConfirmedTopology$ is a set of directed edges (p_i, p_j). An undirected edge is approved if both (p_i, p_j) and (p_j, p_i) appear in $ConfirmedTopology$. Other edges in $ConfirmedTopology$ are said to be suspicious. For each of the proposed implementations, we show in [18] that a polynomial number of paths are used and that they contain $CorrectPaths$. Thus, the sender and the receiver can exchange messages using a polynomial number of paths and message send operations, because each path is of linear length.

Ensuring Safe Message Delivery. We propose a way for the sender and the receiver, that exchange a message over the paths in $getDisjointPaths()$, to stop considering messages and acknowledgments sent by Byzantine nodes. They repeatedly send messages and acknowledgments over the selected vertex-disjoint paths. Before message or acknowledgment delivery, the sender and the receiver expect to receive each message and acknowledgment at least $(capacity \cdot n + 1)$ consecutive times over at least

$f + 1$ vertex independent paths, and by that provide evidence that their messages and acknowledgments were indeed sent by them.

We employ techniques for labeling the messages (in an ARQ style), recording visited path of each message, and counting the number of received message over each path. The sender sends messages to the receiver, and the receiver responds with acknowledgments after these messages are delivered to the application layer. Once the sender receives the acknowledgment, it can fetch the next message that should be sent to the receiver. The difficulty here is to guarantee that the sender and receiver can indeed exchange messages and acknowledgments between them, and stop considering messages and acknowledgments sent by Byzantine nodes.

The sender repeatedly sends message m, which is identified by $m.ARQLabel$, to the receiver over all selected paths. The sender does not stop sending m before it is guaranteed that m was delivered to the application layer of the receiving-side. When the receiver receives the message, the set $m.VisitedPath$ encodes the path along which m was relayed over. Before delivery, the receiver expects to receive m at least $(capacity \cdot n + 1)$ consecutive times from at least $f + 1$ vertex independent paths. Waiting for $(capacity \cdot n + 1)$ consecutive messages on each path, ensures that the receiver gets at least one message which was actually sent recently by the sender. Once the receiver delivers m to the application layer, the receiver starts to repeatedly acknowledge with the label $m.ARQLabel$ over the selected paths (while recording the visited path). The sender expects to receive m's acknowledgment at least $capacity \cdot n + 1$ consecutive times from at least $f + 1$ vertex independent paths before concluding that m was delivered to the application layer of the receiving-side.

Once the receiver delivers a message to the application layer, the receiver starts to repeatedly acknowledge the recently delivered message over the selected paths. In addition, the receiver also restarts its counters and the log of received messages upon a message delivery to the application layer. Similarly the sender count acknowledgments to the current label used, when the sender receives at least $capacity \cdot n + 1$ acknowledgments over $f + 1$ vertex-disjoint paths, the sender fetches the next message from the application layer, changes the label and starts to send the new message.

The Pseudocode of Algorithm 2. In every iteration of the infinite loop, p_i fetches $Message$, prepares $Message$'s label (line 3) and starts sending $Message$ over the selected paths, see the procedure $ByzantineFaultTolerantSend(Message)$. When p_i gets enough acknowledgments for $Message$ (line 4), p_i stops sending the current message and fetches the next. Upon receiving a message msg, node p_i tests msg's destination (line 6). When p_i is not msg's destination, it forwards msg to the next node on msg's intended path, after updating msg's visited path. When p_i is msg's destination, p_i checks msg's type (line 9). When msg's type is *Data*, p_i inserts the message payload and label to the part of the data structure associated with the message source, i.e., the sender, and the message visited path (line 10). In line 12, node p_i checks whether $f + 1$ vertex-disjoint paths relayed the message at least $capacity \cdot n + 1$ times, where $capacity$ is an upper bound on the number of messages in transit over a communication link. If so, p_i delivers the msg to the application layer (line 20), clears the entire data structure and finally sends acknowledgments on the selected paths until a new message is confirmed. Moreover, in line 21 we signal that we are ready to receive a

Algorithm 2. Self-stabilizing Byzantine resilient end-to-end delivery (p_i's code)

Interface: $FetchMessage()$: Gets messages from the upper layer. We denote by $InputMessageQueue$ the unbounded queue of all messages that are to be delivered to the destination;
Interface: $DeliverMessage(Source, Message)$: Deliver an arriving message to the higher layer. We denote by $OutputMessageQueue$ the unbounded queue of all messages that are to be delivered to the higher layer. We assume that it always contains at least the last message inserted to it;
Input: $ConfirmedTopology$: The discovered topology (represented by a directed edge set, see Algorithm 1);
Data Structure: Transport layer messages: $\langle Source, Destination, VisitedPath, IntentedPath, ARQLabel, Type, Payload\rangle$, where $Source$ is the sending node, $Destination$ is the target node, $VisitedPath$ is the actual relay path, $IntentedPath$ is the planned relay path, $ARQLabel$ is the sequence number of the stop-and-wait ARQ protocol, and $Type \in \{Data, ACK\}$ message type, where DATA and ACK are constant;
Variable $Message$: the current message being sent;
Variable $ReceivedMessages[j][Path]$: queue of p_j's messages that were relayed over path $Path$;
Variable $Confirmations[j][Path]$: p_j's acknowledgment queue for messages that were relayed over $Path$;
Variable $label$: the current sequence number of the stop-and-wait ARQ protocol;
Variable $Approved$: A Boolean variable indicating whether $Message$ was accepted at the destination;
Function: $NodeDisjointPaths(S)$: Test S, a set of paths, to encode at least $f + 1$ vertex-disjoint paths;
Function: $FloodedPath(MessageQueue, m)$: Test whether m is encoded by the first $capacity \cdot n + 1$ messages in $MessageQueue$.;
Function: $getDisjointPaths(ReportedTopology, Source, Destination)$: Get a set of vertex-disjoint paths between $Source$ and $Destination$ in the discovered graph, $ReportedTopology$ (Figure 2).;
Function: $ClearQueue(Source)$: Delete all data in $ReceivedMessages[Source][*]$;
Function: $ClearAckQueue(Destination)$: Delete all data in $Confirmations[Destination][*]$;

1 **while true do**
2 $ClearAckQueue(Message.Destination)$
3 $(Message, label) \leftarrow (FetchMessage(), label + 1 \ modulo \ 3)$
4 **while** $Approved =$ **false do** $ByzantineFaultTolerantSend(Message)$

5 **Upon Receive** (msg) **From** p_j;
begin
6 **if** $msg.Destination \neq i$ **then**
7 $msg.VisitedPath \leftarrow msg.VisitedPath \cup \{j\}$
8 **send**(msg) **to next** $(msg.IntendedPath)$
9 **else if** $msg.Type = Data$ **then**
10 $ReceivedMessages[msg.Source][msg.VisitedPath].insert(\langle msg.Payload, msg.ARQLabel \rangle)$
11 **let** $Paths \leftarrow \{Path : FloodedPath(Confirmations[msg.Source][Path], msg)\}$
12 **if** $NodeDisjointPaths(Paths)$ **then**
13 $NewMesssage \leftarrow$ **true**
14 $Confirm(msg.Source, m.ARQLabel, m.Payload)$
15 **else if** $msg.Type = ACK$ **then**
16 **if** $label = msg.ARQLabel$ **then**
 $Confirmations[msg.Source][msg.VisitedPath].insert(\langle msg.Payload, msg.ARQLabel\rangle)$
17 **let** $Paths \leftarrow \{Path : FloodedPath(Confirmations[msg.Source][Path], \langle msg.Payload, msg.ARQLabel \rangle)\}$
18 **if** $NodeDisjointPaths(Paths)$ **then** $Approved \leftarrow$ **true**

19 **Function:** $Confirm(Source, ARQLabel, Payload)$;
begin
20 **if** $CurrentLabel \neq ARQLabel$ **then** $DeliverMessage(Source, Payload)$
21 $(CurrentLabel, NewMessage) \leftarrow (ARQLabel,$ **false**$)$
22 $ClearQueue(Source)$
23 **while** $NewMessage =$ **false do** $ByzantineFaultTolerantSend(\langle Source, ARQLabel, ACK, Payload\rangle)$

24 **Function:** $ByzantineFaultTolerantSend(\langle Destination, ARQLabel, Type, Payload\rangle)$;
begin
25 **let** $Paths \leftarrow getDisjointPaths(ConfirmedTopology, i, Destination)$
26 **foreach** $Path \in Paths$ **do send**$(\langle i, Destination, \emptyset, Path, ARQLabel, Type, Payload\rangle)$ **to first**$(Path)$

new message. When msg's type is ACK, we act almost as when the message is of type $Data$. When the condition in line 18 holds, we signal that the message was confirmed at the receiver by setting $Approved$ to be $true$, in line 18. We note that the code of Algorithm 2 considers only one possible pair of source and destination. A many-source to many-destination version of this algorithm can simply use a separate instantiation of this algorithm for each pair of source and destination.

Correctness Proof. We show that message delivery guarantees hold after a bounded convergence period. The proof is based on the system's ability to relay messages over $f + 1$ correct vertex-disjoint messages (Figure 2), and focuses on showing safe message delivery between the sender and the receiver. After proving that the sender fetches messages infinitely often, we show that within four such fetches, the message delivery guarantees hold; receiver-side delivers all of the sender's messages and just them. The proof in detail appears in [18]. Let us consider messages, m, and their acknowledgements, that arrive at least $(capacity \cdot n + 1)$ times over $f + 1$ vertex-independent paths, to the receiver-side, and respectively the sender-side, with identical payloads and labels. The receiver, and respectively the sender, has the *evidence* that m was *indeed sent by the sender,* and respectively, *acknowledged by the receiver.* The sender and the receiver *clear* their *logs* whenever they have such evidences about m. The proof shows that, after a finite convergence period, the system reaches an execution in which the following events reoccur: (Fetch) the sender clears its log, fetches message m, and sends it to the receiver, (R-Get) the receiver gets the evidence that m was indeed sent by the sender, (Deliver) the receiver clears its log, delivers m, and acknowledge it to the sender, and (S-Get) the sender gets the evidence that m was acknowledged by the receiver. Namely, the system reaches a legal execution.

First we prove that event Fetch occurs infinitely often, in the way of proof by contradiction. Let us assume (towards a contradiction) that the sender fetches message m and then never fetches another message m'. The sender sends m and counts acknowledgments that has m's label. According to the algorithm, the sender can fetch the next message, $m' \neq m$, when it has the evidence that m was indeed acknowledged by the receiver. The receiver acknowledges m's reception when it has the evidence that m was indeed sent by the sender. After nullifying its logs, the receiver repeatedly sends m's acknowledgments until it has evidences for other messages, m', that were indeed sent by the sender after m. By the assumption that the sender never fetches $m' \neq m$, we have that the receiver keeps on acknowledging m until $m' \neq m$ arrives from the sender. Therefore, m arrives from the sender to the receiver, and the receiver acknowledges m to the sender. Thus, a contradiction that the sender never fetches $m' \neq m$.

The rest of the proof shows that (eventually) between every two event of type Fetch, also the events R-Get, Deliver and S-Get occur (and in that order). We show that this is guaranteed within four occurrences of event Fetch. Following the fetch of each of the first three messages and before the next one, the sender must have evidence that the receiver executed event Deliver, i.e., clearing the receiver's log. Note that during convergence, this may surely be false evidence. Just before fetching a new message in event Fetch, the sender must clear its logs and reassign a label value, say, the value is 0. There must be a subsequent fetch with label 1, because, as explained above, event Fetch occurs (infinitely often). Since the sender clears its logs in event of Fetch, from

now on and until the next event Fetch, any corrupted message found in the sender's log must be of Byzantine origin. Therefore, the next time sender gets the evidence that m was acknowledged by the receiver, the receiver has truly done so. Note that between any such two (truthful) acknowledgments (with different labels), say with label, $1, 2, \ldots$, the receiver must execute event Deliver and clean its log, see Algorithm 2, line 22. Since the sender sends over $f+1$ correct paths, and the receiver's logs are clear, eventually the receiver will have evidence for the message with label 0. As corrupted messages originate only from Byzantine nodes and there are at most f such nodes, the receiver's log may not contain evidence for non-sender messages. To conclude, starting from the 4-th message, the receiver will confirm all of the sender's messages, and will not confirm non-sender messages.

5 Extensions and Conclusions

As an extension to this work, we suggest to combine the algorithms for r-neighborhood network discovery and the end-to-end capabilities in order to allow the use of end-to-end message delivery within the r-neighborhoods. These two algorithms can be used by the nodes, under reasonable node density assumptions, for discovering their r-neighborhoods, and, subsequently, extending the scope of their end-to-end capabilities beyond their r-neighborhood, as we describe in the following. We instruct further remote nodes to relay topology information, and in this way collect information on remote neighborhoods. One can consider an algorithm for studying specific remote neighborhoods that are defined, for example, by their geographic region, assuming the usage of GPS inputs; a specific direction and distance from the topology exploring node defines the exploration goal. The algorithm nominates $2f + 1$ nodes in the specific direction to return further information towards the desired direction. The sender uses end-to-end communication to the current $2f + 1$ nodes in the *front* of the current exploration, asks them for their r-neighborhood, and chooses a new set of $2f + 1$ nodes for forming a new front. It then instructs each of the current nodes in the current front to communicate with each node in the chosen new front, to nominate the new front nodes to form the exploration front.

To ensure stabilization, this interactive process of remote information collection should never stop. Whenever the current collection process investigates beyond the closest r-neighborhood, we concurrently start a new collection process in a pipeline fashion. The output is the result of the last finalized collection process. Thus, having a correct output after the first time a complete topology investigation is finalized.

In this work we presented two deterministic, self-stabilizing Byzantine-resilience algorithms for topology discovery and end-to-end message delivery. We have also considered an algorithm for discovering r-neighborhood in polynomial time, communication and space. Lastly, we mentioned a possible extension for exploring and communicating with remote r-neighborhoods using polynomial resources as well.

The obtained end-to-end capabilities can be used for communicating the public keys of parties and establish private keys, in spite of f corrupted nodes that may try to conduct man-in-the-middle attacks, an attack that the classical Public key infrastructure (PKI) does not cope with. Once private keys are established encrypted messages can

be forwarded over any specific $f + 1$ node independent paths, one of which must be Byzantine free. The Byzantine free path will forward the encrypted message to the receiver while all corrupted messages will be discarded. Since our system should be self-stabilizing, the common private secret should be re-established periodically.

References

[1] Ostrovsky, R., Yung, M.: How to withstand mobile virus attacks (extended abstract). In: 10th Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, pp. 51–59 (1991)
[2] Dolev, S.: Self-Stabilization. MIT Press (2000)
[3] Lynch, N.: Distributed Computing. Morgan Kaufmann Publishers (1996)
[4] Al-Azemi, F.M., Karaata, M.H.: Brief announcement: A stabilizing algorithm for finding two edge-disjoint paths in arbitrary graphs. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 433–434. Springer, Heidelberg (2011)
[5] Hadid, R., Karaata, M.H.: An adaptive stabilizing algorithm for finding all disjoint paths in anonymous mesh networks. Comp. Comm. 32(5), 858–866 (2009)
[6] Dubois, S., Masuzawa, T., Tixeuil, S.: Maximum metric spanning tree made byzantine tolerant. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 150–164. Springer, Heidelberg (2011)
[7] Nesterenko, M., Tixeuil, S.: Discovering network topology in the presence of byzantine faults. IEEE Trans. Parallel Distrib. Syst. 20(12), 1777–1789 (2009), see errata via `http://vega.cs.kent.http://vega.cs.kent.edu/~mikhail/Research/topology.errata.html`
[8] Awerbuch, B., Sipser, M.: Dynamic networks are as fast as static networks (preliminary version). In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science (SFCS 1988), pp. 206–220. IEEE Computer Society (1988)
[9] Minsky, Y., Schneider, F.B.: Tolerating malicious gossip. Distributed Computing 16(1), 49–68 (2003)
[10] Li, H.C., Clement, A., Wong, E.L., Napper, J., Roy, I., Alvisi, L., Dahlin, M.: Bar gossip. In: 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, Berkeley, CA, USA, pp. 191–204. USENIX Association (2006)
[11] Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.C.: Gossiping in a multi-channel radio network. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 208–222. Springer, Heidelberg (2007)
[12] Alvisi, L., Doumen, J., Guerraoui, R., Koldehofe, B., Li, H.C., van Renesse, R., Trédan, G.: How robust are gossip-based communication protocols? Operating Systems Review 41(5), 14–18 (2007)
[13] Burmester, M., Le, T.V., Yasinsac, A.: Adaptive gossip protocols: Managing security and redundancy in dense ad hoc networks. Ad Hoc Net. 5(3), 313–323 (2007)
[14] Fernandess, Y., Malkhi, D.: On spreading recommendations via social gossip. In: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2008), pp. 91–97. ACM (2008)

[15] Drabkin, V., Friedman, R., Segal, M.: Efficient Byzantine broadcast in wireless ad-hoc networks. In: Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN 2005), pp. 160–169. IEEE Computer Society (2005), Self-stabilizing Byzantine Resilient 57
[16] Paquette, M., Pelc, A.: Fast broadcasting with byzantine faults. Int. J. Found. Comput. Sci. 17(6), 1423–1440 (2006)
[17] Awerbuch, B., Varghese, G.: Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract). In: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (SFCS 1991), pp. 258–267. IEEE Computer Society (1991)
[18] Dolev, S., Liba, O., Schiller, E.M.: Self-stabilizing Byzantine resilient topology discovery and message delivery. CoRR abs/1208.5620 (2012)